Managing Software Requirements In the Context of the Scientific Enterprise[1]

Steve Larson
Brian Morrison
Jet Propulsion Laboratory
4800 Oak Grove Dr.
Pasadena, CA 91109
818-354-0679
Steve.Larson@jpl.nasa.gov
Brian.Morrison@jpl.nasa.gov

Final Draft
12/14/99
[illegible]

Abstract—Developing production-quality software as part of a scientific research program presents unique challenges. Some of the techniques for managing requirements in this context which have evolved over the last three decades at the Jet Propulsion Laboratory are discussed. The software development effort in support of the Tropospheric Emission Spectrometer (TES) project is presented as an example of the application of these methods.

In certain respects the methods discussed appear to deviate from commonly accepted practice. In an effort to explain the success of these methods, some novel information from the field of cognitive science is explored. We discuss our assumptions about the underlying model of scientific process. The tension between the dualistic view of the world assumed by scientists, and the nondualistic nature of our solutions to the problems of software development is also discussed. We conclude with a discussion of our experience relative to current research into social methods of requirements engineering.

We find that scientists are a unique type of customer, with unique needs from the software development process itself, and not just the end product as many methodologies assume. A social-based approach to development has proven to be the most efficient means of mitigating the difficulties posed by language, educational and cultural barriers.

TABLE OF CONTENTS

1. INTRODUCTION

Since the 1960's NASA's Jet Propulsion Laboratory (JPL) has been involved in developing software systems to support the analysis of data returned by its deep space and Earth-orbiting missions. Operated since inception by the California Institute of Technology, a semi-academic, informal atmosphere has been a hallmark of JPL culture. While many of the Laboratory's successes have been attributed to this culture, Laboratory staff have long recognized the need for discipline in order to ensure mission success.

However, arriving at the right balance of formality vs. unrestrained creativity has often proven to be an elusive goal. Though often achieved by individual projects, this balance has yet to be achieved at an institutional level. An effort in the mid-1980's to codify software practice at the Laboratory resulted in the so-called D-4000 standard [14], roughly based on the military 2167A standard.

The D-4000 standard represented the best practices of the time, but never found complete acceptance. Despite the authors' best efforts to make the standard free of methodological assumptions, it was nevertheless grounded in a mindset of functional decomposition, the waterfall life cycle, and the needs of large software projects. As a result, the standard proved difficult to apply, especially as the software engineering world evolved towards object-oriented methods, rapid prototyping, and evolutionary development. The individual needs of different application domains also presented difficulties for the standard. Over time, groups within the Laboratory responsible for flight software, ground support software, mission control systems, telemetry systems, and science software systems had each evolved their own unique solutions. Although the standard's authors were familiar with these solutions, the final result was widely viewed as non-responsive to the needs of particular domains. In a sense, the standard was a victim of its own success. The authors had done such a good job at abstracting

[1] 0-7803-5846-5/00/$10.00 © 2000 IEEE

the essentials of the software process from the particular instances they were familiar with, that the end result was unrecognizable to many practitioners.

Despite its drawbacks, the D-4000 standard has had a significant impact on the way software developers do their job at JPL. Within the Image Processing and Analysis Section, a tailored form of the standard was applied to the development of the Multimission Image Processing System, and later, to the science software projects supporting the Earth Observing System (EOS) instrument projects at JPL. The Tropospheric Emission Spectrometer (TES) project is the most recent of the latter class of software project within the section.

While the D-4000 standard provided a formal framework for software development, the local methods of developing software that had evolved over the previous twenty years prior to D-4000 remained in use. These methods have never been formally captured, and if they are written down at all, are scattered across a broad range of project internal documentation. Probably the most influential methods used were never written down at all, forming an important, though tacit body of knowledge about how to succeed at science software projects. This paper is an attempt to describe some of the previously tacit knowledge embedded in the local corporate culture.

The immediate sources of many of the methods and approaches used by the TES team were earlier EOS, and EOS-related Earth remote sensing projects. These include the Airborne Visible/Infrared Imaging Spectrometer (AVIRIS), Atmospheric Emission Spectrometer (AES), Multiangle Imaging Spectroradiometer (MISR), Advanced Spaceborne Thermal Emission Spectrometer (ASTER), and the Atmospheric Infrared Sounder (AIRS). These projects represent a distinct thread of methodological development within the Section, largely under the influence of Graham Bothwell, who was responsible for managing all of them except AIRS for at least part of the project lifetime. The general approach to science software discussed here is deeply indebted to Graham's insights and guidance over the past decade.

2. Challenges Posed by Science Projects

TES-Specific Challenges

The TES project is a somewhat extreme example of science software development projects at JPL. With the possible exception of the MISR project, TES stands out as unique in the difficulties posed by the instrument system and the science. TES is a Fourier transform spectrometer designed to measure high resolution infrared spectra from low earth orbit. It is an unprecedented measurement system with its unique combination of spectral resolution, signal to noise, and ability to operate in a nadir and limb viewing mode. The purpose of the mission is to provide a three-dimensional map of tropospheric ozone and its chemical precursors, plus temperature, water vapor and surface parameters. These data are to be used by global climate researchers to validate models of the Earth's climate system.

The prototypical JPL instrument is an imaging system. This is especially true of the systems supported within the Image Processing Section. One of the challenges posed by TES is the unfamiliar nature of the measurement. As a spectrometer with only sixteen spatial channels but over 5,000 spectral channels, the data do not lend themselves easily to visual interpretation. As an interferometer, the data require complex mathematical algorithms to convert raw bits into physical units. These methods, too, were new to the software team.

Processing of the measured spectra into vertical species profiles involves a highly complex algorithm that challenges the software development team's mathematical and physical reasoning skills. The nature of the processing problem is such that it is impossible to calculate a deterministically correct solution. One is in the position of having to accept what is best characterized as the most *likely* solution. The techniques for doing so are highly sophisticated, representing the cutting edge of optimization techniques. The non-deterministic nature of the algorithm lends an uncertainty to the process that some developers find discomfiting.

Most importantly, the science algorithms for processing the data were largely undefined at the start of the project. Developing the algorithms has required a multi-year effort on the part of a geographically distributed science team. The algorithm development process is still ongoing, and is expected to continue well after launch, currently scheduled for December 2002. The requirements management aspect of the software development is thus intimately coupled to a scientific research program. In a very real sense, our requirements specifications job is synonymous with the algorithm research program.

General Challenges of Science Software Projects

We present below what must be regarded as a partial discussion of the challenges posed by science software development projects in general. We focus on developing a fundamental understanding based on the epistemological characteristics of the scientific enterprise. Certain factors relating to the basic needs of scientists are also discussed.

Nomology—In what will become a recurring theme of this paper, we begin by observing that there are fundamental differences between the development of instrumentation, scientific theories, and software. Our first observation is that these activities do not share a common nomology (rules of reasoning). This fact is a source of difficulty in developing requirements for science software projects.

Instrumentation always obeys the laws of physics. When reasoning about instruments, the laws of physics are clearly the appropriate rules to apply. As an activity directed towards correctly understanding and interpreting physical phenomena, science is clearly situated in the cognitive realm of human functioning. But what nomology applies here?

Scientists will often assert that the same rules of physics (or whatever specialty the project involves) apply to scientific activities as well. However, this is only part of the story. Physical reasoning plays a primary role, but mathematical thought, deductive and inductive logic apply, as do a host of progressively less "scientific" factors, including the preconceptions of each scientist as to the types of potential solutions they will seek, and more personal prejudices unrelated to science *per se*. At this point, the question of what rules of reasoning apply to the conduct of scientific investigations becomes confounded with the question of what rules apply to human behavior in general. This question is the province of cognitive science.

The entire project of cognitive science may be viewed as an effort to uncover the nomology of the mental world. In the half-century since it emerged as a recognizable research program[2], various avenues in linguistics, philosophy, anthropology and neuroscience have been explored. In spite of the many fascinating theories that have been developed about human cognition, there appears to be no nomology equal in stature to (for example) physics, to provide a suitable foundation for understanding cognitive phenomena.

How we interpret the failure of this pursuit has important implications for how we approach science software problems. If we interpret the situation in the modernist tradition, and view it as merely a matter of time before the rules are uncovered and explained, then we are likely to misinterpret some (but not all) of the difficulties encountered in science software projects. In particular, difficulties arising out of the preliminary and contingent nature of scientific knowledge, as discussed in greater detail later, may be misunderstood as difficulties that may be addressed simply by working harder to clarify understanding or answer questions. The situation is often not that simple.

On the other hand, if we adopt a more post-modern view we risk a profound conflict with the foundation of science. Descartes, who may be regarded as the founder of the modernist faith in an objective, external reality regarded it as the scandal of philosophy that no one had yet successfully refuted the view that nothing exists outside one's subjective reality. Heidegger[3] [11] inverts this formulation, saying, "the 'scandal of philosophy' is not that this proof has yet to be given, but that *such proofs are expected and attempted again and again*." Although Heidegger's challenge does not appear to threaten in any way the ability of science to continue to make strides in understanding the world, it strikes closer to home with software engineering, a field that finds itself hard-pressed to ground its conclusions in the "real world". The Heideggerian challenge implies that the rules we seek may be inherently undiscoverable.

The inability of software engineers to ground the wisdom of their trade in something equivalent to the laws of physics hampers their ability to communicate with a science team. The problem boils down to the inability to provide a convincing justification for software engineering methods. Orthodox scientific training leads scientists to expect and accept as true only statements based on "scientific" principles and evidence. Although considerable work has been done to establish the value of practices such as peer reviews, diagramming, and documentation, the studies which support them appear too anecdotal to be accepted uncritically by the scientific mind, and their interpretation lacks the supporting body of theory that would enable one to trace conclusions back to first principles. Lacking such a foundation, it can be difficult to communicate to a science team the value of requirements engineering.

It is therefore especially important to avoid justifying the effort devoted to requirements on the basis of the software engineering profession itself. We have found it imperative that the development team formulate requirements issues explicitly in terms of the scientific objectives of the science team.

Methodology—A scientist, frustrated with the inscrutable and slightly suspect methodology of an engineering team, might naively insist that the proper way of developing science software is the way they themselves have always done so. Typically, this means one or two people writing a code for their own use, or for the use of their immediate work group. These codes may grow and evolve as an integral part of a scientist's research, often over decades.

For a scientist accustomed to working with computers in this mode, it can be very difficult to adjust to any other way of working. Confronted with the prospect of being deprived of their former intimacy with the inner workings of a program, and their total control over its features, a scientist may resist the efforts of a software engineer to apply their own methods to the software development.

This resistance may be explicit or implicit. In some cases, application of change management techniques (e.g., [16]) may be effective. However, we have found it generally impossible, and counterproductive, to try to convince scientists used to working directly on code to give it up entirely. Programming is often inextricably embedded in a scientist's working methods. We favor the development of team structures that provide programming support to scientists who require it, and an organizational bridge to join these activities to the more formal software development activities (cf our discussion of prototyping below).

A scientist who has successfully written many software programs in their career is likely never to have written a

[2] See [15] for a general overview of the history of cognitive science.

[3] As quoted in [4].

requirement, as the issue of communicating them rarely arose. If it did, was most likely with a colleague or student, who could reasonably be expected to share much of their understanding of the domain. Thus, with some justification, scientists will often challenge the need to write requirements.

Though it is possible to mandate documentation of requirements programmatically, this approach is not likely to elicit the kind of cooperation necessary to produce a quality requirements specification. To address this issue we again advocate the development of effective team structures that facilitate requirements development with a minimum of intrusion on the science team's normal activities.

Key to success in this area is the presence on the team of individuals who have sufficient training in the scientific domain to act as bridges between the science team and the more computer science-oriented software engineers. Given the highly technical nature of most science software requirements, we have found that Master's and doctoral level degrees in scientific disciplines are often required. A person in this type of position must be sufficiently interested in the science that they devote the time and energy needed to understand it, but not so interested that they cannot disengage themselves from the science to focus on software-specific matters. These are the individuals who are best capable of eliciting tacit requirements, and correctly interpreting explicitly given requirements. As we shall discuss more fully later, we find the issue of the interpretation of requirements to be fundamental to science software development.

How many such "bridges" are needed to be successful? It is difficult to provide an exact formula. A scientist-bridge ratio of between 1:1 to 2:1 seems appropriate. For example, the TES project currently has four full-time science algorithm developers on the science team resident at JPL, and the equivalent of two to three full-time algorithm developers off site. We have three people who serve in bridge roles at JPL, and one off site. We can see a clear need for at least one additional person in a bridge position.

Communications—One must pay special attention to the communications skills of the software engineering staff. Although staffing may be regarded as a project management issue rather than a requirements management issue, the importance of communications skills cannot be overlooked in a discussion of science software requirements engineering, as the cultural and vocabulary barriers are often substantial.

A scientist able to communicate effectively with other scientists may be less effective when confronted by non-scientists. Weakness in communications skills may be masked in scientist-scientist interactions by the shared language of the discipline, and a willingness to tolerate those weaknesses on the basis of mutual regard. A similar situation obtains with regard to engineers. Training in both disciplines places little emphasis on interpersonal skills. Indeed, it is the attitude of many scientists and engineers that these skills are simply not important as long as the quality of their technical work is good.

Since one typically cannot influence the make-up of a science team, it is essential that care be taken when selecting the staff who will be expected to carry out the requirements engineering tasks.

Two Cultures, One Shared Purpose—Establishing a shared purpose in a heterogeneous scientists/engineering team is more difficult than in a homogeneous case. At least part of the problem stems from the type of goals each group is culturally predisposed to establish. Science may be characterized as an activity that is larger than any single participant. Its scope is generally taken to span centuries and nations, and its goals explicitly formulated in terms of what is currently beyond reach. Engineering groups, on the other hand, typically have a more near-term, immediate focus.

Physicist and writer C.P. Snow described the gulf separating scientific and engineering culture 40 years ago [16] as follows:

> Pure scientists and engineers often totally misunderstand each other. Their behavior tends to be very different: engineers have to live in a very organised community, and however odd they are underneath they manage to present a disciplined face to the world. Not so pure scientists. [Scientists] have statistically a higher proportion in politics left of centre than any other profession: not so engineers, who are conservative almost to a man.

Snow was critical of the attitudes of contemporary scientists. He lamented that, " Pure scientists have by and large been dim-witted about engineers and applied science. They couldn't get interested. They wouldn't recognise that many of the problems [of engineering] were as intellectually exacting as pure problems, and that many of the solutions were as satisfying and beautiful." Snow went further to say that, "We prided ourselves that the science we were doing could not, in any conceivable circumstances, have any practical use. The more firmly one could make that claim, the more superior one felt."

Although much has changed, especially in terms of the attitude of scientists towards the relevance of their work to society at large, since Snow wrote those words, the polarization between science and engineering culture remains a significant force today. It is also important to recognize that the situation is not one-sided. Engineering culture has its own ingrained attitudes that can make it difficult for them to appreciate the needs of scientific research. One must also be cautious about applying generalizations to specific individuals. Nevertheless, cultural differences pose one of the largest barriers to effective science team/engineering team partnerships. Mitigating those differences requires a constant effort throughout the project life cycle.

Given these differences, how does one formulate a shared vision that both sides will buy into? One approach we have had success is to seek an expression of the team's objectives that lends itself to a satisfactory interpretation from either perspective. For example, on the TES project, our overarching goal is to deliver a global data set of vertical species profiles, plus temperature and surface parameters, of ozone and it precursors. This formulation is preferable over one that emphasizes science, such as "to do ozone research", or software, such as "to develop and deliver the profile retrieval software".

The value of culturally neutral expressions extends to the development of requirements. Wherever possible, science software requirements should be written in such a way as to preserve their scientific intent. A requirement which has been formalized to the point where the scientific context is no longer evident will be incomprehensible to a scientist, and will likely convince them that the process is either irrelevant, or even harmful to their goals. It will also be a substantial barrier to verification[4], as the scientist is unlikely to be willing to verify a requirement that they perceive as irrelevant.

3. Some Shortcomings of Existing Methods

The discussion to follow must be set against a background of appreciation for the value of traditional software engineering techniques. Even the waterfall model, maligned as is may be in the literature and in practice, contains an essential kernel of truth about the basic order of events in software development. The TES team's approach to documentation and standards for reviews are based on the D-4000 standard, tailored to meet project needs. We challenge here the value of certain techniques or assumptions specifically in the context of science software development. We begin with a criticism of the way in which some problems in requirements engineering are posed.

Posing the Problem for Solution

Dorfman and Thayer [1] provide a comprehensive list of requirements management issues that appear to us both comprehensive in scope, and typical in their expression. An abbreviated version of their list is as follows:

1) Inability of engineers to write software specifications.

2) Management desire to emphasize code & test at expense of requirements.

3) Lack of customer cooperation in requirements verification.

4) Lack of customer understanding of purpose of requirements.

5) Tool/methodology selection.

6) Effective lack of knowledge that system requirements are essential to software requirements.

7) Lack of training in requirements allocation from system to software.

8) Habit of management in placing people with little software experience in charge of large software projects.

This list should be familiar to all experienced software engineers. Different writers will express it in different forms, with differing emphases, but the message is fairly constant. Science software projects are the same as all others insofar as they share in these issues to one degree or another. Although we discuss Dorfman and Thayer's formulation here, we view it as an example of the way in which software development issues are commonly framed. No specific criticism of Dorfman and Thayer is intended.

Our difficulty is with the language in which the issues are expressed, especially items 3 and 4. By identifying the customer's "lack of cooperation" or "lack of understanding" as an issue, the problem is posed in terms of a supposedly correctable fault in the customer, and suggests that solutions lie in the area of educating the customer, or providing them with encouragement or incentives to become more involved.

Unmet Needs—We propose to reformulate the problem in terms of unmet needs. In the case of a scientist customer, we believe those needs are profound and may even have a physiological basis. If our physiological hypothesis is correct, failure to meet those needs may incur resistance that cannot be reasonably overcome or redirected for the convenience of software developers.

We must emphasize at the outset that the discussion to follow is not intended to be scientifically rigorous. Our intent is to provoke discussion of how physiological aspects of the human organism may bear upon the behavior of people involved in the software development process, and how that bearing might influence our formulation of software engineering problems.

As background, we present now some information on how the human brain is thought to function. Ornstein and Sobel [9] present an intriguing account of the effects of over and understimulation on the brain. Studies of brain development in rats[5] have shown that stimulation directly affects the growth of neurons. Conversely, understimulation results in neural atrophy. The ability to induce growth extends into old

[4] The subject of customer involvement in requirements verification is discussed in more detail below.

[5] See discussion of the work of Marion Diamond in [9], p. 222.

age, where as little as a week of appropriate stimulus is needed to produce measurable changes.

The brain regulates the amount of information flow it receives, similar to the homeostatic functions that regulate blood pressure and body temperature. "Compared with one less experienced or developed, the more experienced brain does receive less sensory information because we have learned to extract out just those bits of information we need... At each step we need less and less, we need attempt less and less. As we age we get more and more needful of strong stimuli.[6]"

The profound impact of mental understimulation on the overall health of the individual is dramatically illustrated by the results of studies that suggest a link to cancer[7]. "When the information deficit reaches some critical value, the brain sends a nonspecific signal ... indicating the need for novelty or information; carcinogenesis is the body's mode of providing 'information-novelty' which is subsequently fed back to the brain and which attempts to rectify the relative information underload signaled by the brain."[10]

Our thesis here is simply this: that the information needs of a scientist (or any other customer type with a strongly intellectual component to their profession) are deeply rooted in their physiology. We do not suggest that it is possible for a software team to endanger the health of a scientist merely because the latter finds software boring. A bored scientist will disregard the software process and return to their work long before any real health hazard could materialize.

To illustrate the importance of our physiological hypothesis, the following analogy may be helpful. Asking a scientist to leave the interesting and challenging world of their research for a day is akin to asking them to skip lunch. They may experience some discomfort, but if the perceived gains are sufficient, the sacrifice may be acceptable. Failing to meet their information needs over the long term, however, is akin to asking them to fast for a year.

Scientists have chosen a profession that meets their prodigious need for intellectual stimulation, and their chosen field is probably the most important means of fulfilling this need. It is our experience that boredom is a frequent complaint of scientists expected to participate in software development activities. These complaints disappear when the focus of the activity is sufficiently close to their own concerns that they can participate as scientists rather than as conscripts.

Our discussion of the physiology of information needs adds another dimension to the problem of how to successfully engage a scientist customer in the requirements engineering process. As is commonly understood in the software engineering community, failure to understand requirements adequately is likely to lead to project failure, or at the very least to cost and schedule overruns, and to a disgruntled customer unlikely to provide your organization with repeat business.

Writing Specifications

Considerable work has been done on defining the attributes of good requirements. Typically, the list of desiderata includes such features as completeness, correctness, and absence of ambiguity. Despite the fact that recent methodologies emphasizing iterative development and rapid prototyping have challenged the practicality of achieving these qualities in an absolute sense, especially at the beginning of a project, they have not been abandoned outright. As discussed below, we find the nature of science projects introduces new complications into the task of writing "good" specifications.

The root cause of the trouble lies in the fact that software plays an intimate role in the process of acquiring knowledge. In this capacity a science software project is unique among software projects in general. How one approaches the task of software development in this context is highly dependent on one's underlying model of the scientific process. Equally critical is how one resolves the ambiguities intrinsic to the process. We begin with a discussion of our assumed model of the scientific process, and proceed to explore the complications arise as a result of weaknesses in those assumptions.

Model of the Scientific Process—An inadequate model of the scientific process can lead to unrealistic expectations of the process, and of the requirements engineering activities which depend on that process. Despite its popularity, we find Kuhn's notion of paradigm shifts [7] an unsatisfactory basis for understanding the dynamics of science software projects. In our experience Kuhn's model does not provide an accurate description of the way science is carried out on a day-to-day basis[8], and it fails to provide a basis for a rational understanding of the scientific process.

The model assumed in the present work is based on Lakatos' [3] notion of a research program. A research program is more than a specific plan for conducting research. It encompasses a broad range of assumptions and attitudes concerning which questions should be asked, how they should be formulated, the form of solution one should seek, the methods by which those solutions are developed, and the ultimate goals and value of science. Earth system science is an example of a research program.

At the level of how scientists carry out their work day-to-day, we adopt as a model what Lakatos calls *sophisticated*

[6] Ornstein and Sobel [9], p. 214-215.

[7] The quote which follows is excerpted from Ornstein and Sobel [9], p. 226.

[8] L. Pierce Williams [2] criticized Kuhn and others for attempting to construct a philosophy of science upon the basis of a history of science that was in itself insufficient to support such a project.

methodological falsificationism. In this model, a scientific theory T is falsified if and only if another theory T′ has been proposed with the following characteristics: (1) T′ has excess empirical content over T: that is, it predicts novel facts, improbable in the light of, or even forbidden by T; (2) T′ explains the previous success of T, that is, all of the unrefuted content of T is contained (within the limits of observational error) in the content of T′; and (3) some of the excess content of T′ is corroborated.

Sophisticated falsificationism attempts to moderate the extremism of naive falsificationism by establishing the conditions under which a theory could be legitimately saved from potentially falsifying observations by the addition of auxiliary hypotheses or theories. According to sophisticated falsification no experiment, experimental report, observation statement or well-corroborated low-level falsifying hypothesis alone can lead to falsification. There is no falsification before the emergence of a better theory.

This latter point is important, as it sheds light on the methodological reasons why science software development can be as complex as it often is. It is usually necessary for the scientist to maintain several competing theories simultaneously, and to modify them continually in light of new observations or related theoretical developments. Scientific theories typically constitute the most important (from the customer's point of view) component of a requirement specification.

Dualism—A dualistic relationship between the mind and an objectively knowable world is implicit in both Kuhn's and Lakatos' understanding of science. For present purposes, we may take the "mind" to represent the scientist who attempts to understand the physical world. In this model, the mind is separate from, and independent of, the physical world. Another common assumption of the dualistic viewpoint is that knowledge consists of internal representations of the world, representations that are evaluated according to the degree to which they correspond with the observed state of affairs in the world.

It is ironic that perhaps the strongest challenge to the representational theory of knowledge comes from science itself. A classic work on vision in frogs by Humberto Maturana, et al, [5] showed that certain nerve fibers in the frog's retina were able to detect patterns such as that a fly might make, and that triggering these nerve fibers lead to behavior appropriate to catching the fly in that spot. In Winograd and Flores' [4] discussion of this work, they conclude that, "the frog with optic fibers responding to small moving dark spots does not have a *representation*[9] of flies." If Maturana's finding is correct, that it is structural patterns in neurons rather than a literal representation of objects in the world that underlie the mind's activity, then what exactly are we doing when we engage in model building? In the structuralist view of mental functioning, we are modifying connections between neurons, but nowhere do we ever create something that could be construed as a model.

If we regard mental models as an emergent property of the neural structure of the brain, we have removed it one step from the so-called "objective" world, which we suppose it represents. What now may be said of the connection between our models and "reality"? The post-modern tradition in philosophy has raised this issue in many different forms. The essential issues (with respect to the present subject) that emerge in their work concern the effects of previous understanding.

The Hermeneutic Circle and Measurement Systems

We designate the total system of instrumentation, processing software and supporting physical theory a *measurement system.* A measurement system is neither completely physical nor completely cognitive in nature. The *hermeneutic circle* is a concept drawn from textual analysis. Originally applied to the problem of interpreting texts, the concept has since been expanded and applied elsewhere. We apply it here to the problem of acquiring and understanding scientific data (text) about the physical world.

Simply put, the hermeneutic circle says that we cannot understand a text except in terms of other texts with which we are already familiar. It is understood as a barrier to understanding an author's "real" intent versus applying one's own interpretation based on one's own prejudice as the "meaning" of the text.

A scientist presupposes that "nature" is the "author" of the text (data) he or she is attempting to understand. In our version of the hermeneutic circle, the scientist's status as "reader" of this text is ambiguous. As described in greater detail below, the scientist plays a role in the authorship of the data, and thus is in danger of reading as nature's hand what they themselves have written. This is presented as a problem for requirements engineering, as it affects their principle source of requirements.

Instrumentation and Preunderstanding—We begin by considering the status of physical instrumentation. Simplistically, we may regard the objective world to be synonymous with the physical world, and we may regard instrumentation systems as part of that world, with the cognitive processes associated with theory building and software located in the mental or cognitive realm.

Though simplistic, this basic distinction provides valuable insight into the software development process. As an aside, we believe this distinction is basic to all software development projects, and helps explain why methodologies based on experience with hardware system development have proven inadequate for software projects.

[9] Emphasis in the original text, Winograd and Flores [4], p. 46.

In response to the post-modern challenge to dualism, we modify this simplistic view by recognizing a continuum between physical instrumentation and interpretive activities. By virtue of the interaction of thought (in the form of observational theories, as defined below) and physical reality, the line between the two may become confounded.

Since Galileo, science has relied upon instrumentation to extend the reach of human senses. Instrumentation is an artifact of the physical world. As such, it must obey the laws of physics. In fact, we may say that physical instrumentation obeys the laws of physics infallibly, regardless of whether or not the instrument is correctly designed or constructed, or makes useful measurements at all. But instruments are designed by human beings—scientists—and those designs are based upon prior physical theory regarding the physical world.

The theories used in the construction of instruments belong to a class of theories Lakatos called *observational* theories [4]. Observational theories are physical theories that are regarded by practicing scientists as sufficiently well proven to be regarded as unproblematic. They are distinguished from *explanatory* theories, which are theories intended to explain some observable phenomenon. Experiments are intended to test explanatory theories, and are carried out on the basis of observational theories.

In a science software project, it is important to recognize first, that a theory may function as an observational theory in one context, and as explanatory in another. Observational theories must always start out as explanatory theories. New discoveries may appear to falsify an observational theory, and cause its assumed explanatory power to be called into question. Although no practicing scientist thinks of what they do in exactly those terms, in practice they are keenly aware of the possibility of this occurring.

On the TES project, an example of an observational theory would be the body of theory underlying the algorithm we call the "forward model". This algorithm describes how a mathematical representation of an absorption/emission spectrum can be constructed based on knowledge of the atmospheric state, the laws of electromagnetic radiation, and certain quantum mechanical properties of the molecules present in the atmosphere. Prior to its being accepted as a satisfactory basis for use in analyzing a measurement, the theory had prove its ability to explain real spectra.

As is implied by our model of falsificationism, no such proof is ever completely conclusive. Indeed, there are inevitably approximations made in solving complex equations and an absence of critical measurements to completely validate the theory. On TES, we regard the forward model theory as fairly well established, but there remain open questions about its ability to model aerosols, and scattering, and the underlying laboratory measurements used in the calculation are known to be incomplete and in some cases inaccurate. Where laboratory data are simply too poor, theoretical calculations provide the needed parameters. These factors condition our view of the correctness and completeness of our forward model requirements. Since the software requirements express a physical theory, they are subject to the same vicissitudes as the theory itself.

Although the characterization is somewhat dramatic, one could say that science software requirements are best understood as set of educated guesses embedded in a much larger network of guesses. There are no certainties, and there is no way to insulate your requirements against unexpected outside influences beyond the project team's control. Indeed, scientists expect to be occasionally jostled from previously comfortable conclusions, and a science software requirements engineer should have similar expectations.

As discussed above, an instrument always obeys the laws of physics, and it obeys explicitly only those physical laws of which the designers were aware. The choice of instrument, the details of its design, the situation in which it is placed, and the uses to which it is put are all the result of deliberate choices made by the scientist. Thus, an instrument, while patently an artifact of the physical world, is always an expression of the preunderstanding of the scientist who uses it. It may not be regarded as independent of the theoretical understanding of the scientist responsible for it's design and specification.

The Hermeneutic Circle and Software—The hermeneutic circle becomes even more problematic in the area of software development. The case of the Total Ozone Mapping Spectrometer (TOMS) on the NIMBUS-7 spacecraft illustrates the situation well. NIMBUS-7 launched in October 1978, and the TOMS began returning data in November of that year. The instrument returned data until the instrument finally failed in May 1993.

The TOMS science team was responsible for the software used to process TOMS data to produce ozone maps. Based on then-current understanding of Antarctic ozone, the team put a trigger in their software to detect low ozone amounts, and to flag the data as defective.

Although evidence of the ozone hole was present in TOMS data from 1979 onward, the team overlooked the presence of the hole until 1985 when a team lead by Joseph Farman reported it. A reanalysis of the TOMS data revealed the full extent of the ozone hole, which Farman's ground-based measurements had been unable to do, but nevertheless, the team missed out on being able to claim the discovery of this important atmospheric feature.

It is clear that the hermeneutic circle is a real and important feature of the science software landscape. When considering the possible effects of the hermeneutic circle on a science software development project, it is essential that one does not attempt to find ways to bypass it altogether. As Heidegger asserts, "if we see this circle as a vicious one and look out for ways of avoiding it, even if we just sense it as an inevitable imperfection, then the art of understanding has been misunderstood from the ground up."[11, p. 194]

While it remains an essential part of any scientific activity to work assiduously to uncover all of one's hidden assumptions and prejudices, the hermeneutic circle precludes that one's understanding will ever be complete or totally objective. Current trends in the software industry towards "good enough" software implicitly recognize this fact. However, even if one constructs a process (as we have) whereby requirements are determined to be "good enough", even then there remains some uncertainty. How do you know they are good enough? Are you telling yourself what you want to hear?

Problems with Specification Languages—To address the problems of ambiguity inherent in natural languages, some authors[10] suggest using a requirement specification language, or structured English, to express requirements. However, we find this approach to be fraught with difficulties, and find that the scientific process itself provides what may be the best available means of dealing with the issues of ambiguity. In our discussion of the latter point, we begin to show more clearly why we believe the social aspects of science software requirements engineering are most important.

The linguist Noam Chomsky posited a meta level of expression—simple, declarative sentences which, through a set of well-defined rules could be transformed into any possible expression. Indeed, some compiler systems are designed with exactly this model. Chomsky's theory provides us with a prototypical basis for requirements specification languages. There are two key assumptions one must make to adopt this model: first, that there is a meaning independent of any particular expression, and second, that nothing is lost in the translation. Post-modernist thinkers have challenged both of these notions. We find that the challenge has been effective, at least in the case of science software requirements.

A trial use of structured English [12] was firmly rejected by the customer on the AES project. The resulting prose was found to be too much like reading computer code, which some members of the project team, the project manager and principle investigator in particular, found virtually impossible to read. The same individuals are involved in the same capacity on the TES project, and the use of structured English was not proposed for TES.

In addition to problems of customer acceptance, we find other difficulties with specification languages on science software projects. In discussing the advantages of informality Goguen [8] refers to the *efficiency* of language. By efficiency is meant the ability of natural language to have meaning that is context dependent, and to have multiple meanings simultaneously. While these are the very "problem" features specification languages are meant to solve, Goguen argues that it would be difficult, and perhaps impossible, to communicate without the ambiguity in natural language.

We believe that specification languages cannot completely solve the problems they set out to solve by virtue of the fact that they are themselves constructed from natural languages. That they can never be free from the supposed problems of natural language may be demonstrated by an example[11] taken from Hatley and Pirbhai [12] (see figure 1).

The specification in this example would appear to be clear, and unambiguous, but exactly what is a *pulse*? This word and the phrase "shaft rotation" illustrate the impossibility of constructing a specification language without reference to objects in the real world. But real-world objects have names that are defined in terms of natural languages, and there is no way to extricate them from that context.

```
PSPEC 1; Measure Motion

For each pulse of shaft rotation:

        add 1 to DISTANCE COUNT
        then set:

                DISTANCE = DISTANCE COUNT/
                           MILE COUNT
```

Figure 1 Example of Structured English

One assumption of advocates of specification languages is the assumption that whatever ambiguities inherent in the natural language expression are artifacts of the natural language itself, and not the thought of the writer.

Can a requirements language say what we mean? One of our top-level science requirements is to "...incorporate a flexible list of additional species..." The document does not specify what "flexible" means, or exactly what "incorporate" means (incorporate into the code? a database? what?). Nevertheless, this requirement represents the intent of the science team, and does not constitute a problem for the development team. Both the science team and the developers share an understanding of this requirement, despite its lack of specificity. More importantly, both teams recognize that there remain some open issues as to exactly how this requirement should be met, and further understand the process by which the issues are to be resolved. We treat our requirements as *items of information*, as Goguen [8] defines them: "an *item of information* [is] an interpretation of a

[10] See, for example, Hatley and Pirbhai [12].

[11] The example PSPEC is taken from p. 53 in [12].

configuration of signs for which a social group can be held accountable." The social group here is the project team, comprising scientists and software developers.

The efficiency of language is a key part of our ability to deal with imprecise knowledge. Attempting to remove the ambiguity in the specification would prove to be disruptive to the flow of work, and if attempted as an exercise in its own right would likely meet with the derision of the science team.

Advocates of the use of requirements languages may argue that the important thing is the exercise of attempting to perform the translation. That is, it is the process of attempting the translation that produces the most important results in the form of improved understanding. Although this argument has some merit, it is insufficient to justify such as approach in our case. We must make it clear that we do resolve the ambiguities. However, this effort is part of the science process rather than a separate exercise.

Resolving Ambiguities—Ambiguities are a common feature of our specifications. However, we do not simply leave them be, and permit the inevitable consequences to ensue. In the case cited above, the ambiguity is deliberate on the part of the science team. They do not know enough about what they really need to be more precise.

The science team and project team members responsible for the particular subsystem the requirement pertains to meet formally once a week, and informally every day. These are explicitly the forum for resolving the ambiguity of the requirement. However, even in this context the focus is not on how to state the requirement more clearly (although this does occasionally occur), it is most typically in the context of discussing potential design solutions.

Placing the requirements clarification process in this context helps to bring the discussion closer to the actual world of concern of the science team. Relative to the discussion above of boredom and physiology, this is an example of how the process is made a more integral part of the scientist's work. It is the job of the software engineers to recognize that the process is addressing ambiguities in requirements, and to modify the specifications accordingly.

If the ambiguities in a natural language specification are due to the ambiguities in the understanding of the writer, then the use of specification languages can only help uncover their existence. But in the instance of a science requirement, an ambiguity in a requirement often reflects a genuine lack of clear understanding of how the world works.

Is the imposition of a specification language the best means of uncovering this lack of understanding? We believe not. Scientists have developed a highly sophisticated means of performing this function. It does not work perfectly, but the process of peer review, both formal and informal, is an effective means of discovering areas of weak understanding. Since this process is already intrinsic to the practice of science, the salutary effect of specifications languages appears redundant.

Software peer reviews—It is not completely satisfactory to rely on the peer review process of scientists alone. On TES, we extend this process into the software development activity through informal peer reviews of software work products. The science team routinely participates in these reviews. These reviews provide further social context for clarifying, reworking and understanding requirements. It is this highly interactive exchange of ideas and interpretations that we believe gives the greatest value to the software peer review process.

On the TES project we use the peer review process as one technique to better understand science requirements. This technique is familiar ground to the scientist and represents a fairly common mechanism to attempt to better understand science goals and objectives by emphasizing what is known and to expose areas of weak understanding. The key to developing requirements during the peer review process is to understand that the science requirements themselves are evolving in parallel to the scientific research. Through peer reviews we embed the development of requirements with the process of advancing the research by working collectively (scientist and software developer) to better understand the goals and objectives of the science.

So the process is not one of gained requirement content simply through the transcription of information, gathered and written into itemized requirements. Rather, it is the process of frequent interaction using diagrams, drawings and words to express a science requirement in ways comfortable to both parties that help better define the requirements.

If We Knew the Requirements, We'd Be Done—What turns out to be more problematic than uncovering areas of weak understanding is the problem of managing what is already known to be either poorly understood, or altogether unknown. Scientists are usually quick to point out the limits of their knowledge. Indeed, the notion of intellectual honesty requires that they make this known, lest they appear to claim knowledge they do not have.

In the scientific investigations we have been associated with, the science team is usually confronted by a bewildering array of unanswered questions, each vying for priority. There are always far more questions to be answered than there are scientists to answer them. In challenging science projects, the science team often faces significant difficulties simply keeping track of what it must solve in order to progress scientifically, let alone write software requirements.

From this perspective, the scientist may claim (and many do) that if they knew what the requirements were, the problem would be solved, but solving the problem is what they have been asked to do! The situation is not as circular as it may appear, but it underscores the unique difficulty of specifying requirements for a science project. In a sense, the requirements represent knowledge of the solution, and all of the programming and testing that go on in system development are merely a means for verifying the requirements!

We regret that we have not yet found an effective means of managing the "known unknowns" in our systems.

A Comment on Emerging Social Approaches to Requirement Engineering

Despite our enthusiasm for social approaches to requirements engineering problems, we are less certain about the promise of attempts to apply ethnographical techniques to requirements engineering. For reasons that will become clearer further on, the use of ethnography, as reported in [13] seems to create more problems than it solves. Our basic objection is that the injection of an ethnographer into the requirements engineering process merely introduces another interdisciplinary communications problem, which is one of the principle problems we find in science software development projects. We are also concerned about the reception this technique would meet with if applied to a science project. A physical scientist may have difficulty accepting the involvement of a social scientist (even a proxy), and may be prejudiced against the methods and conclusions of the ethnographer in such a way as to negate the potential benefits of the approach.

4. Role of Scientist-Developed Codes

As discussed above, many scientists are accustomed to writing their own codes in support of their individual activities. Although this can present some difficulties to a software engineering team, certain opportunities arise as well. We begin with a discussion of how scientists use models to document their knowledge, the problems associated with those models, and the emerging role of software as a form of publication.

Nature of scientific models—Scientists model their understanding of the physical world in many ways. Some means of expression are more-or less universal, such as equations, and x-y plots. Others are domain-specific, such as the rules for drawing cladistic relations in tree form, conventions for visually representing molecules and notation used to express group-theoretical statements. Others are simply ad hoc, such as the figures used to express the geometry of a physical arrangement, or the sources and sinks of carbon in the biogeochemical cycles.

The writer of a science software requirements specification is presented with several difficulties. First, there is the bewildering array of modeling methods used by scientists. Second is the lack of a clear and unambiguous means of linking all of the artifacts a scientist will present as reference documentation. Typically, the ability to reliably negotiate the maze of equations, papers and data in a particular field comprises a portion of the training a scientist receives when they earn their Ph.D. Clearly, it is unreasonable to expect that a software engineer would retrace all of these steps simply in order to write a specification.

The solution we have found to be most effective is to employ people with scientific training, preferably at the Masters or doctoral level, to serve as de facto interpreters. These are not the same people who write the software requirements, that job is reserved for experienced software engineers.

The key aspect of their role is that, while possessing the intellectual tools to understand the science, they are not personally committed to performing the science, and therefore are capable of standing apart from the science activity in a way the science team members cannot. They are able to answer questions about the software requirements that the science team cannot, without necessarily having all of the knowledge needed to answer a scientists question about the science.

Another important aspect of the use of models in expressing scientific theories, which must eventually find their way into a software specification, is their nature as models. Recall that the dualistic concept of truth is based on mental representations and their correspondence to observables. This correspondence is taken for granted in science. On the other hand, the lack of objects appropriate for (especially visual) modeling is a well-known problem in software engineering. Modeling has established its value in software engineering despite its overt lack of direct correspondence to anything in the "real world". It might even be said that the methodologies associated with modeling techniques arose in response to this deficiency. For the developer of science software, the critical question is what is the relationship of the model a software engineer builds of the software and the models the scientist builds to express their understanding of the world?

Software as Publication—One way to begin to answer this question is to explore the evolving role software plays in science. With the emergence of networked computer systems, scientists have enjoyed an ever-increasing ability to communicate and share data. The role of the Internet in the development of string theory is a case in point, where the importance of informal electronic communications challenges the relevance of traditional peer-reviewed journals. Within NASA's Earth Observing System (EOS) program, some scientists have begun to recognize that making data available on the internet is in some ways equivalent to publishing a paper.

Although we have not heard it explicitly discussed in scientific circles, we expect it is only a matter of time before scientists recognize the status of a software program as an executable expression of scientific theory. In the context of the present discussion, it is productive to assume this viewpoint.

Doing so changes the nature of the question of the relationship between scientific and software models. The software program as model is introduced as a third form of expression that both frustrates traditional software engineering approaches, while clarifying in a way no other approach can, the intent of the scientist.

On the TES project, we regard software developed by scientists as prototype code. It is generally given in conjunction with other, more traditional expositions of the underlying scientific theory, and is typically developed by the scientist in order to verify that the theory works correctly. One issue for the software developer is which form of expression takes priority? The textual description of the theory, with its associated figures and equations, is generally the form of expression used as the basis for discussion among the science team, and usually communicates the underlying ideas in more general fashion. However, it is only the code form that has successfully negotiated the translation process into software, and only this form that has been tested empirically.

When the scientific problems are complex, as they are in the case of the TES retrieval algorithm, it is insufficient to provide a purely theoretical justification for a particular analysis technique. There are simply too many subtleties to permit anything but an empirical demonstration of the correctness of the underlying theory to justify its acceptance.

This situation raises the immediate concern of software engineers. What the scientist wants, with justification, is the ability to write the code before the system is designed.

On the TES project, we have embarked on a long algorithm prototyping effort. From the scientists' perspective, the purpose of this effort is to develop the processing algorithms. In the terms used in this paper, we would say they are developing the observational theory that will underlie the data the team delivers to the larger scientific community. Prior to its assuming a status of being an observational theory, it must be shown to be a satisfactory explanatory theory with regard to observed measurements of the type TES will make. It is this process of transition from explanatory status to observational status, and the possibility of it operating in reverse, that is the source of so much difficulty in the system development.

From the perspective of the software developers, the purpose of the prototype is to establish an understanding of what the code must do, i.e., the requirements. But is it satisfactory to regard a prototype code as an expression of requirements? Even if it were true, in what way could a prototype code be interpreted as a requirement?

It would seem easy to say that the problem may be solved by regarding the prototype code as a by-product of a requirements definition activity, with the software engineers taking what they have learned from it and codifying it in a requirements document. However, this is not sufficient. Even after a long and in-depth prototyping activity such as TES', the software engineers may not be in possession of sufficient understanding of the domain to write an acceptable specification.

Given the ongoing nature of the scientific process, the prototype can never be regarded as complete until it has been definitively replaced with another code. That means that interest in the performance and functionality of the prototype will continue throughout the lifetime of the project, and possibly even afterwards. This is especially true if the deliverable code does not contain all of the functionality included in the prototype. A prototype code may contain a number of alternative methods, and false starts that later turn out to be of interest.

A prototype code is also of interest in it role as arbiter of truth. A code that is taken as validating a scientific theory then becomes a "reference code" against which other codes are measured. If another code cannot come up with the same answer, or prove why the original code is wrong, then it is regarded as wrong. The prototype code developed by the TES project is expected to play this role.

How can a prototype code be considered an expression of software requirements? We have taken the approach that a prototype can be considered part of a software requirement only insofar as it is used as a reference code, i.e., as a source of results that any subsequent code must match. Even here one must be careful to specify the exact test cases to be used, as it is possible for two codes to get identical results on one case, and diverge on another. The code itself is maintained under configuration control, and is provided to the software developer as a detailed description of how the requirement might be satisfied, but the particular solution is not levied as a requirement.

On the TES project, the prototyping is treated as a major part of the requirements definition process. An entire segment of our workflow is devoted to managing the prototyping process, and controlling the flow of new requirements form the prototype to the production code. The science team controls the prototyping process almost entirely. Individual science team members propose new ways of processing the data. Following a process outside the software realm, they decide which algorithms merit inclusion in the prototype. At this point the engineering staff become involved by estimating the resources necessary to implement the changes, and the teams jointly schedule the new work. Once completed, the science team evaluates the changes. If they are found to be acceptable the whole process begins over again, this time in the context of the production system development. In this new context, the newly validated science algorithm is treated as a new requirement.

The prototype code becomes the reference code for testing, as well as a detailed example of how the requirement might be satisfied. As part of the science evaluation process, the scientist who initiated the process must provide written documentation of the algorithm as well as specifications for test cases and test results. These are entered into configuration control along with the prototype code. The whole is then associated together in one change package in the configuration management system.

5. Conclusions

Science software development presents unique challenges to the software engineering profession. These challenges derive from the basic epistemological problems of science. Working scientists are aware of these problems, but rarely attempt to formulate them explicitly in terms of epistemology. We have presented some philosophical and cognitive information that provides a more explicit means of describing the challenges of acquiring knowledge. In this light, the problems of science software development are cast in terms that emphasize the importance of team structure, domain understanding and the development of common interpretations of requirements versus a concentration on how formally to express those requirements.

6. Acknowledgements

The authors would like to acknowledge Ken Scott, Annette Larson and Robert Vargo for their valuable assistance in reviewing early drafts of this work. The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

7. References

[1] Richard H. Thayer and Merlin Dorfman, Eds. *Software Requirements Engineering*, 2nd Ed., Los Alamitos, CA: IEEE Computer Society Press, 1997.

[2] L. Pierce Williams "Normal Science, Scientific Revolutions and the History of Science", in Imre Lakatos and xxx Musgrave, Eds., *Criticism and the Growth of Knowledge*, Cambridge: Cambridge University Press, 1970.

[3] Imre Lakatos, "Falsification and the Methodology of Scientific Research Programmes", in Imre Lakatos and xxx Musgrave, Eds., *Criticism and the Growth of Knowledge*, Cambridge: Cambridge University Press, 1970.

[4] Terry Winograd and Fernando Flores, Understanding Computers and Cognition, New York: Addison Wesley, 1988.

[5] Humberto Maturana, "Biology of Language: The Epistemology of Reality", in G.A. Miller and E. Lenneberg (Eds.), *Psychology and Biology of Language and Thought: Essays in Honor of Eric Lenneberg*, New York: Academic Press, 1978.

[6] Jamed Siddiqi and M. Chandra Shekaran, "Requirements Engineering: The Emerging Wisdom," *IEEE Software* **13**, 15-19, March 1996.

[7] Thomas S. Kuhn, *The Structure of Scientific Revolutions*, Chicago: University of Chicago Press, 1962.

[8] Joseph A. Goguen, "Formality and Informality in Requirements Engineering," Proceedings of the Second International Conference on Requirements Engineering, Los Alamitos: IEEE Computer Society Press, 1996.

[9] Robert Orenstein and David Sobel, *The Healing Brain: Breakthrough Discoveries About How the Brain Keeps Us Healthy*, Cambridge, MA: Malor Books, 1987.

[10] Augustin de le Pena, *The Psychobiology of Cancer*, New York: Praeger Publishers, 1983.

[11] Martin Heidegger, *Being and Time*, (translated by John Macquarrie and Edward Robinson), New York: Harper & Row, 1962.

[12] Derek Hatley and Imtiaz Pirbhai, *Strategies for Real-Time System Specification*, New York: Dorset House, 1987.

[13] Stephen Viller and Ian Sommerville, "Social Analysis in the Requirements Engineering Process: From Ethnography to Method", in *Proceedings: 4th IEEE Syposium on Requirements Engineering*, Los Alamitos, CA: IEEE Computer Society Press, 1999.

[14] *Software Management Standard*, JPL D-4000, Pasadena, CA: Jet Propulsion Laboratory.

[15] Howard Gardener, *The Mind's New Science: A History of the Cognitive Revolution*,

[16] Charles Percy Snow, *The Two Cultures and the Scientific Revolution*, New York: Cambridge University Press, 1959.

Steve Larson is a software project manager at the Jet Propulsion Laboratory. He has been involved in the development of science data processing systems for several NASA Earth remote sensing projects. Prior to joining JPL, he worked as a research assistant in the areas of plasma physics and low-energy nuclear physics. He has an A.B. in Art from Occidental College and a M.S. in Physics from California State University, Northridge.

Brian Morrison is a software system engineer at the Jet Propulsion Laboratory. He has been involved in the development of avionic software systems at Lockheed Aeronautical Systems and has worked on the command and control software for several NASA deep space tracking stations. Currently, he is involved in the development of a science data processing system as part of NASA's earth

remote sensing program. Mr. Morrison has a B.S. in Computer Science from California State Polytechnic University and an MBA specializing in project development from the University of La Verne.